7 Средства программирования в среде Unix

7.1 Общая характеристика среды программирования

Тезис «Unix – среда программирования» (ср.: Керниган Б.В., Пайк Р. Unix – универсальная среда программирования. [4]) Гипербола, но в целом сама концепция, заложенная в ОС Unix, предполагает поддержку полного цикла создания и модификации ПО, включая системное ПО вплоть до компонентов системы. Однако это не противоречит существованию более удобных и функциональных инструментов, составляющих более мощную среду (что также соответствует концепции).

Основные компоненты среды программирования:

– Редактор(ы) исходных текстов

– Компилятор(ы)

– Средства управления выполнением (обработкой) проекта

– Средства отладки и др. дополнительные инструменты

– Библиотеки

– Средства управления файлами проекта

Т.н. ***toolchain*** – основной набор инструментов, обеспечивающий получение исполняемой программы из исходных текстов. Остальные компоненты обеспечивают эффективное применение toolchain.

Система действительно предоставляет практически все необходимые компоненты: стандартные средства работы с текстами, компилятор, отладчик, библиотеки, утилита **`make`** естественным образом присутствуют в ней, а shell играет роль своего рода «IDE».
Полнота набора зависит от конкретного дистрибутива и построенной ее конфигурации.

Далее рассматривается в первую очередь случай программирования на C/C++, но то же в основном применимо к большинству «универсальных» языков (и с оговорками – не только компилируемого типа).

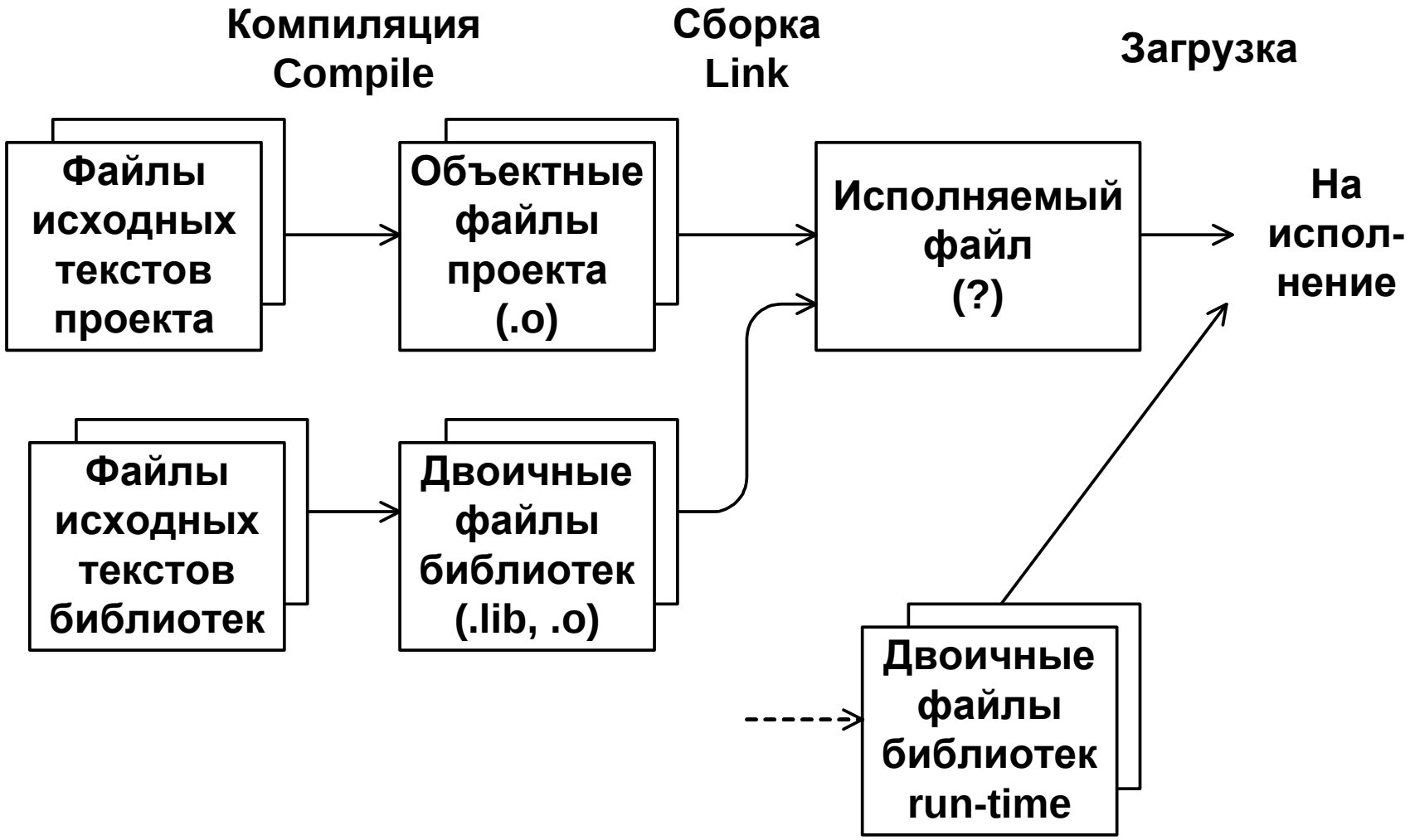


Схема типичной технологической цепочки для проекта

7.2 Редакторы исходных текстов

Фактически уже рассмотрены. Теоретически может использоваться любой доступный редактор, но на практике не все редакторы одинаково удобны.

Примеры:
– **`vi`**/**`vim`**
– **`nano`**
– **`emacs`**
– **`Geany`**
и т.д.

Некоторые редакторы можно рассматривать как сильно ограниченную по возможностям IDE: распознавание и обработка синтаксиса редактируемого исходного текста, вызов внешних инструментов. В частности, такие возможности заложены даже в **`vim`**.

7.3 Компилятор

Задача – преобразование (трансляция) исходного текста программы в двоичный (промежуточный) код, в том числе с присоединением подключаемых файлов (не двоичных).

В Unix традиционно присутствует:
– **cc** – ***C C***ompiler,
– **gcc** – первоначально ***G***NU ***C C***ompiler, позже ***G***NU ***C***ompiler ***C***ollection

Современный **gcc** – фактически надстройка над набором (suit) компиляторов различных языков, за счет чего и обеспечивается их поддержка. К поддерживаемым языкам относят: C/C++, Objective-C/C++, Fortran, Ada, D, Go, BRIG (HSAIL)

Например:
`gcc` – различает исходные файлы **`.c`** и **`.cpp`**, считая их язык C или C++ соответственно
`g++` – непосредственно GNU C++ compiler, ожидает исходных файлов только на C++
`gcc -xc++ -lstdc++ -shared-libgcc` – вызов **`gcc`**, эквивалентный прямому вызову **`g++`**

На отдельных специфических платформах **`gcc`** может оставаться и просто GNU C Compiler.

Как минимум для C/C++ программ **`gcc`** обеспечивает весь цикл преобразования от исходного текста до исполняемого кода, но с помощью опций можно потребовать только отдельные этапы или специфические дополнительные функции.

Простейший случай – компиляция и сборка двух файлов:

gcc *mysrc_main.c mysrc_sub.c*

Построенный исполняемый файл получает имя **a.out**.
Промежуточные объектные файлы не сохраняются.

gcc имеет большое количество параметров и опций. Несколько наиболее употребимых:

Опция	**Эффект**
-o	имя выходного файла (обычно исполняемого или библиотеки)
-l	использовать заданную библиотеку
-c	только компиляция исходного текста, без сборки
-x	явное указание языка входных файлов

Опция	**Эффект**
`-w`	управление предупреждениями («уровнем чувствительности»)
`-d`	включение отладочной информации в исполняемый файл
`-pipe`	использовать pipe вместо временных файлов

и так далее.

Теперь предыдущий пример можно расписать постадийно:

1) Компиляция исходных текстов в объектные файлы:

```
gcc -c mysrc_main.c
gcc -c mysrc_sub.c
```

2) Сборка исполняемого файла **`myprog`** из объектных:

```
gcc -o myprog mysrc_main.o mysrc_sub.o
```

7.4 Управление обработкой проекта (make и makefile)

Сложность и разнообразие вызовов компиляторов и других компонентов, необходимость учитывать взаимосвязи файлов, зависимость от внешних параметров и т.п. приводят к необходимости автоматизировать процесс обработки проекта. (Здесь и далее ***проект*** – взаимосвязанные файлы, возможно разнотипные. Проект не обязательно только программный.)

Простейшее решение – сценарий (сценарии) командного интерпретатора, где учтены все требуемые варианты. Решение работоспособное, но трудоемкое для пользователя.

Специализированное средство – утилита **`make`** (и аналоги).

Использование **`make`** или аналогичного инструмента – практически общепринятый подход, в т.ч. в MS Visual Studio.

Основные задачи **`make`**:
– выполнение различных операций над проектом по требованию пользователя (программиста)
– автоматическое построение «маршрута» выполнения операции
– автоматический контроль актуальности файлов проекта

Специализированный язык сценариев **`make`**. Для сценариев (скриптов) традиционно принято имя **`makefile`** или **`Makefile`**

Язык **`make`** – декларативный: описывается требуемый результат и ведущие к нему зависимости, но не точный алгоритм обработки.

Вызов **`make`**:

Выполнение сценария из файла с именем **`makefile`**, первая по порядку цель:

```
make
```

Выполнение сценария из явно указанного файла с именем **`mymakefile`**, первая по порядку цель:

```
make -f mymakefile
```

Выполнение сценария **`makefile`**, явно указанная цель:

```
make mytarget
```

Общая структура **`makefile`** – последовательность ***правил*** (***rule***) Порядок следования правил в большинстве случаев значения не имеет:

```
правило 1 …
правило 2 …
…
```

Символ «#» – признак комментария, часть строки после него игнорируется.

Структура правила (роль «логических скобок» играют отступы – табуляции или пробелы):

```
цель : зависимость зависимость …
  команды
```

Здесь ***команды*** – обычные команды shell со всеми его возможностями

Трактовка правил:
– достижение цели через последовательность команд
– цель может достигаться после выполнения зависимостей

Две категории ***целей*** (***target***):
– цели-***файлы*** – считаются достигнутыми, если файл существует и актуален (не старше своих зависимостей)
– ***абстрактные*** цели («метки»).

Зависимости (***dependency***) – условия достижимости целей.

Достигнутая цель может быть зависимостью для следующей цели.

Получив определенную цель, **`make`** самостоятельно выстраивает дерево зависимостей и проходит по всем путям, которые необходимы чтобы вернуть проект в актуальное состояние.

Пример простейшего сценария make для двухфайлового проекта:

```
myprog : main.o subfunc.o
  gcc main.o subfunc.o –o myprog
main.o : main.cpp
  gcc –c main.cpp
subfunc.o : subfunc.cpp
  gcc –c subfunc.cpp
clean:
  rm *.o
```

Здесь все цели, кроме **`clean`**, – файлы; **`clean`** – абстрактная цель.

Нередко абстрактные цели добавляются для улучшения читаемости сценариев и упрощения их последующего редактирования:

```
build : myprog
  …
myprog : main.o subfunc.o
  …
```

Очевидно, описание большого сложного проекта такими средствами достаточно трудоемко.
Решению проблемы служат расширения синтаксиса **make**.

Переменные:

Тип – строка (строковый литерал)

Объявление переменной:

имя=строка

Обращение к переменной:

… $*имя* …

Предопределенные переменные (предварительное объявление не требуется, поддержка автоматическая):

Имя	**Использование**
$@	имя цели
$^	список зависимостей
$<	имя первой по счету зависимости

Ряд имен переменных считаются «общепринятыми», например:

Имя	**Использование**
CC	компилятор для исходных текстов на C, обычно **gcc**
CXX	компилятор для исходных текстов C++, обычно **g++**
CFLAGS	опции компилятора C
CXXFLAGS	опции компилятора C++
LINK	«сборщик» объектных модулей и библиотек (linker)
LFLAGS	опции компоновщика
INCPATH	пути к заголовочным файлам
LIBS	список подключаемых библиотек (путей к ним)

и так далее

Шаблонные правила:

```
.<суффикс_файлов_depend> .<суффикс_файлов_target>
    ... #команды
```

Например, компиляция в объектные файлы всех файлов исходных текстов ***.c**:

```
.c .o :
        gcc -c $^
```

Очевидно, выигрыш от шаблонных правил быстро растет по мере увеличения количества файлов в проекте.

Примеры

Сценарий без оптимизации (проект под ОС Raspbian Linux, Raspberry Pi):

```
UARTSender : UART.o Comm.o main.o
	gcc -Wl,-O1 -lpthread -o UARTSender UART.o
Comm.o main.o

UART.o : UART.cpp
	gcc -pipe -O2 -I. -I../Libs -c UART.cpp

Comm.o : Comm.cpp
	gcc -pipe -O2 -I. -I../Libs -c Comm.cpp

main.o : main.cpp
	gcc -pipe -O2 -I. -I../Libs -c main.cpp

clean :
	 rm *.o UARTSender
```

Видны неоднократно повторяющиеся строки с почти одинаковым содержимым, многократные повторы одних и тех же имен. При добавлении файлов в проект их обработку придется дописывать заново. При изменении опций команд придется редактировать их во всем сценарии. Напрашивается подстановка с параметризацией.

Аналогичный по действию сценарий с использованием переменных:

```
TARGET = UARTSender
CC     = gcc
CXX    = g++
CFLAGS = -pipe -O2 -Wall -W -D_REENTRANT -fPIC
CXXFLAGS = -pipe -O2 -Wall -W -D_REENTRANT -fPIC
INCPATH  = -I. -I../Libs
LINK   = g++
LFLAGS = -Wl,-O1
LIBS   = -lpthread

$(TARGET) : UART.o Comm.o main.o
	$(LINK) $(LFLAGS) $(LIBS) -o $@ $^

UART.o : UART.cpp
	$(CXX) $(CXXFLAGS) $(INCPATH) -c $^
```

```
Comm.o : Comm.cpp
	$(CXX) $(CXXFLAGS) $(INCPATH) -c $^
main.o : main.cpp
	$(CXX) $(CXXFLAGS) $(INCPATH) -c $^
clean :
	rm *.o $(TARGET)
```

Очевидного выигрыша в размере сценария нет, но сопровождать его будет проще: локализовано управление опциями, можно расширить их набор, не усложняя сценарий.

Сценарий с использованием шаблонных правил:

```
TARGET = UARTSender
CC     = gcc
CXX    = g++
CFLAGS = -pipe -O2 -Wall -W -D_REENTRANT -fPIC
CXXFLAGS = -pipe -O2 -Wall -W -D_REENTRANT -fPIC
INCPATH  = -I. -I../Libs
LINK   = g++
LFLAGS = -Wl,-O1
LIBS   = -lpthread

$(TARGET) : UART.o Comm.o main.o
      $(LINK) $(LFLAGS) $(LIBS) -o $@ $^

.cpp. .o :
      $(CXX) $(CXXFLAGS) $(INCPATH) -c $^ #все *.cpp
```

```
.c .o :
        $(CC) $(CFLAGS) $(INCPATH) -c $^ #все *.c
clean :
        rm *.o $(TARGET)
```

Два типа файлов-зависимостей для одного типа файлов-целей распознается как коллизия, но сценарий выполняется корректно.

Развитые IDE обычно имеют собственный формат описания проекта, а **`makefile`** генерируется каждый раз заново как сценарий «сборки».

7.5 Отладка и дополнительные инструменты

Утилиты общего назначения, используемые в том числе и для анализа и корректировок файлов программ:

`grep` – поиск строки (подстроки) в файле (с использованием регулярных выражений)

`diff` – построчное сравнение файлов, формирование файла с описанием различий

`patch` – «применение» описания различий («патча») к исходному файлу для приведения его в соответствие с «эталонным»

`ldd` – список зависимостей исполняемого файла от внешних библиотек

и т.д.

db, **gdb** – отладчики

Стандартные системные отладчики позволяют выполнять базовые операции над отлаживаемым процессом: установить точку останова, выполнить инструкцию, перейти на адрес, вывести дамп и т.п.

Более функциональные и специализированные отладчики: **truss**, **ktrace**, **Total View** и др.

Основа для реализации отладки – специализированные библиотеки, например **PTrace** и **CTrace** (дополнительная с улучшенной поддержкой многопоточности).

Пример – внедрение «отладочного» кода в программу

7.6 Библиотеки

Библиотеки – понятие, определение.

Подходы к построению библиотек:
– «обертки» других функций (в т.ч. системных вызовов);
– полная реализация функционала внутри библиотеки;
– частичная реализация внутри библиотеки с обращениями к внешним функциям (в т.ч. к системным вызовам)

Стандартные библиотеки – рассматриваются как часть системы, описываются спецификациями Unix-систем:

1) Библиотеки ***системных вызовов*** (***syscall***) – «обертки» функций ядра:

– «полностью стандартные» – базовый IPC, единый для различных семейств и версий Unix-систем: **`fork()`**, **`exec()`**, большинство вызовов IPC, файловый ввод-вывод (через дескрипторы **`fd`**) и т.п.

– «не полностью стандартные»

2) Библиотеки ***буферизованного ввода-вывода*** – надстройка в адресном пространстве прикладных процессов над системным вводом-выводом, собраны в **`stdio.lib`** (заголовочный файл **`stdio.h`**). Фактически часть стандарта языка С.

3) Другие системные вызовы и комбинированные реализации (надстройки над системными вызовами), например работа с датой и временем.

Дополнительные библиотеки (все прочие) – могут быть прикладными и системными, переносимыми и машинно-зависимыми, и т.д. Часть описывается в ANSI-стандарте C.

Некоторые системные вызовы могут не иметь соответствующих «оберток» в библиотеках (имеющиеся библиотеки отстают от актуального ядра), даже если эти вызовы описываются в справке **`man`**. Для доступа к таким вызовам служит «универсальная» оберточная функция **`syscall()`**:

```
int syscall( int call_number, ...);
```

Она обеспечивает обращение к системному вызову, заданному его номером, с передачей параметров в виде переменного списка, результат предполагается целочисленным.

Номера вызовов зависят от семейства и версии ОС (ядра ОС), и их значения включаются в SDK как символические константы.

Пример:

В стандартном SDK обычно отсутствует (по крайней мере, до недавнего времени) обертка для системного вызова **`gettid()`** (получение ID потока), и его приходится выполнять «вручную», используя символическую константу **`SYS_gettid`**:

```
int tid = syscall( SYS_gettid);
```

При желании можно добавить для удобства собственную реализацию «обертки»:

```
int gettid( void) { syscall( SYS_gettid); }
```

или даже в виде макроса:

```
#define gettid() ((pid_t)syscall(SYS_gettid))
```

Сложно объяснить, почему такая простая реализация не включена в SDK, хотя и константа **`SYS_gettid`**, и большинство других функций для работы с потоками там присутствуют.

Типичные проблемы при использовании библиотек:

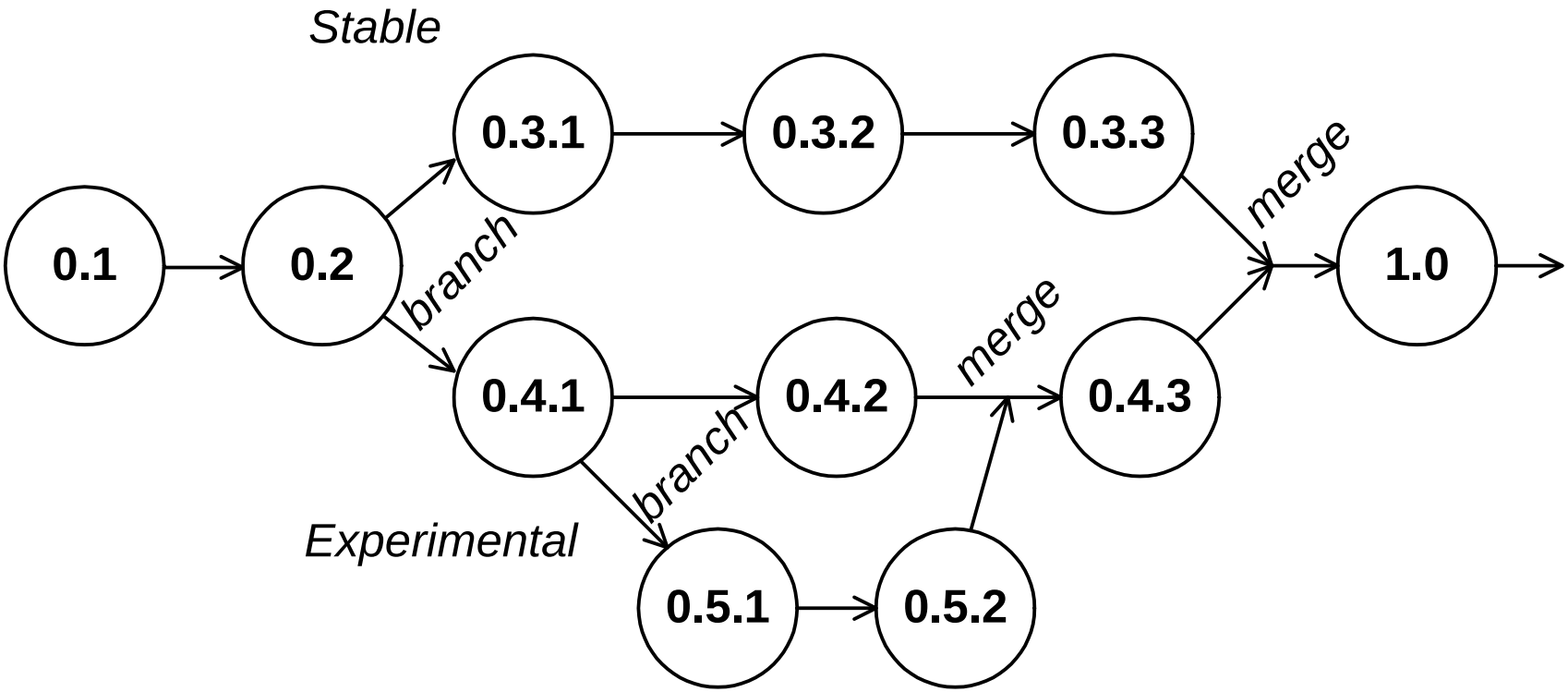
Неполная совместимость, различия в поведении библиотек различных производителей (если библиотека предназначена для подмены существующей) или различных версий.
Нет единого исчерпывающего решения. Частично – средства централизованного управления инсталляцией системы, компонентов и библиотеку, но это противоречит открытости и гибкости замены ПО.
Аналогичная по сути проблема в Windows-системах – т.н. «DLL Hell»

Соответствие двоичного файла библиотеки правилам (требованиям) головной программы (также адекватность описания его в текстовом заголовочном файле).

Различие низкоуровневых форматов представления данных и их согласование.
Например, числа с плавающей точкой: **`float`** (32-разр.) или **`double`** (64-разр.) в C/C++, **`REAL`** (48-разр.) в Pascal.

7.7 Управление проектами

Проблема: параллельная модификация файлов проекта несколькими разработчиками (особенно при распределенной работе); одновременное существование нескольких версий одних и тех же файлов; необходимость поддержания целостности проекта, надежного его хранения, обеспечения коллективного доступа.



Версии проекта

Задача сложная и трудоемкая, особенно выявление различий между файлами. Необходима автоматизация.

CVS (Concurrent Versions System) – наиболее удачная ранняя программа, собирательное название для подобных программных решений.

Другие продукты: SVN (Subversion control system), SourceSafe, ClearCase, Mercury, GIT и т.д., различающиеся по подходам, возможностям, технической реализации и проч.

7.8 Специализированные IDE

В целом не играют роль настолько доминирующую, как в MS Windows. В ряде случаев являются кросс-платформенными.

Примеры:

– Qt (IDE Qt Creator с системой библиотек)

– Code::Blocks (IDE, компилятор и др. инструменты)

– Lazarus (Delphi-подобный IDE)

– Eclipse (IDE, платформа для построения IDE, в первую очередь для Java, но также C/C++/C#, PHP, Perl, Python, Ruby и т.д.)

и т.д.

Как правило, сохраняется принцип отделения IDE от toolchain (и от средств управления проектами) – возможность различных сочетаний и независимой замены.
В некоторых случаях такая гибкость сильно ограничена – например, с конкретными специфическими аппаратными платформами могут быть совместимы лишь конкретные библиотеки и/или их версии, которые, в свою очередь, сочетаются с конкретной версией IDE.

7.9 Элементы «хорошего стиля» программирования

(Т.н. «***Tao of the Unix Programming***», первоначальное авторство Eric S. Raymond)

1. Правило ***модульности***: простые компоненты с понятными интерфейсами.

2. Правило ***ясности***: ясность лучше изщренности.

3. Правило ***соединения***: программы изначально рассчитываются на взаимодействие.

4. Правило ***разделения***: отделение функциональных «слоёв» (алгоритмы – механизмы, интерфейсы – «движки», и т.д.)

5. Правило ***простоты***: так просто, как возможно; сложные конструкции допустимы только в случае необходимости.

6. Правило ***умеренности***: так коротко, как возможно (например, простые сценарии вместо большой полнофункциональной программы.

7. Правило ***прозрачности***: наглядность для упрощения ревизии и отладки программ

8. Правило ***надёжности***: надежность как следствие ясности и простоты. (Однако, иногда сложный алгоритм бывает намного лучше простейшего, если, конечно, он отлажен.)

9. Правило ***представления***: представление знаний в данных.

10. Правило ***наименьшего удивления***: предсказуемость интерфейсов и других решений.

11. Правило ***тишины***: минимизация выходных сообщений.

12. Правило ***восстановления***: максимально заметное извещение об ошибках.

13. Правило ***экономии***: экономия времени программиста за счет времени машины.

14. Правило ***генерации***: программная генерация программ вместо ручного кодирования.

15. Правило ***оптимизации***: работоспособность важнее эффективности, оптимизация после появления рабочего прототипа.

16. Правило ***многообразия***: универсальные решения часто бывают менее эффективны; единственно верные решения подозрительны.

17. Правило ***расширяемости***: заложенные при проектировании возможности масштабирования, наращивания функциональности и т.д.